Actor-based Parsing

Héctor Jiménez
Facultad de Ciencias Físico-Matemáticas
Universidad Autónoma de Puebla
hjimenez@fcfm.buap.mx

Guillermo Morales
Centro de Investigación y
Estudios Avanzados del IPN
gmorales@cs.cinvestav.mx

July, 1997

Abstract

We present the SymAct language, an actor-based language oriented to describe the interaction between grammar symbols. The language manages communication among text pieces and is based in the parallel logic language Concurrent Prolog and the chart parsing method. This paper exposes several examples of programming and its compiled code.

1 Introduction

The large volumes of natural languages text requiere to take advantage of new architectures and to utilize the recent approaches based on concurrent models. There has been proposals to parse sentences in parallel logic languages [6] and to face the syntactic ambiguity problem [10]. At the present paper we undertake the idea of minimal model of distributed computation to provide a parallel approach to parsing. Parallel logic programming contrasts with a pure definition of actor model [1]. Yet, it has been shown by several experimental language implementations and applications [5, 3, 4] that there exists some common features of actor model and parallel logic languages. This nearness offer resources to other application language paradigm. Furthermore, if it is requiered parsing sentences we should select a method with good performance. Shift-reduce parsing is the candidate to adapt in the logic programming paradigm. In the actor model each actor has a mail address, holds only one mailbox and every computational step it reads its mailbox to perform some primitive action [11]. However, to deal with grammar symbols we have to change some features of the actors. Every one symbol will communicate with its neighbors therefore, it will have two mailboxes. From these ideas we propose the language SymAct which compiles symbol behaviour into Concurrent Prolog. In the next section it is stated the fundamental issues of shift-reduce parsing and of parallel logic languages. Then is presented the SymAct language as message-passing between grammar symbols and several examples are discussed. Finally in the conclusion section we analyze some improvements to the language and what contexts are appropiated to apply this approach.

2 Background

2.1 Shift-Reduce Parsing

Perhaps shift-reduce [2] parsing is the most adequated method for analyzing sentences of a language. This is due to its linear complexity and its power on a large set of grammars. Nevertheless, when we facing the ambiguity, it is necessary to extend the syntactic analysis with new devices just like the generalized parsing [10]. Let us remember some relevant aspects of shift-reduce parsing:

1. If we have a state with $[A \to \beta.]$ and the current symbol u is in Follow(A), then reduce the preceding string to A.
2. If we have a state with $[A \to \beta_1.\beta_2]$ and the current symbol is in First(β_2) then shift the current symbol.

Generalized parsing adds a stack of multiple paths through the automata representing transitions between states of the parsing.
It is important to adapt these methods to new approaches. We will take the parallel logic programming as our base for integrate such methods.

2.2 Parallel logic programming elements

Let us remember some basic elements of parallel logic languages. There are several parallel logic languages [9]. We are basing our work on Concurrent Prolog. Parlog, GHC and concurrent Prolog execute a process collection through the description of each one of these as *clauses*:

$$H \leftarrow G | B_1, \ldots, B_k$$

where H is the name of the process and B_i $(1 \leq i \leq k)$ are the processes which are executeed whenever the *guard* G (normally a test) is satisfied. Commonly in parallel logic languages the synchronization and communication mechanism is realized by a restriction on the access to logical variables. Mainly the kind of synchronization mechanism characterizes to each language. In Concurrent Prolog is used the 'read-only-variable' which is a special presentation of any logical variable prefixed by the `?` symbol: a process is restricted to consult the variable only when it has been instantiated, forcing to the process suspend until the variable is instantiated.

3 A language for message-passing between grammar symbols

Basically we have integrated the idea of actor [1] with chart parsing [8] and the implementation of logical grammar in Parlog, like Matsumoto's technique.

3.1 Principles of design

Let us consider the basis of shift-reduce and the possibility of communication through channels between grammar symbols. If there are conditions of a reduce action then we must replace the RHS's symbols by the rule non terminal symbol. For each rule, we select a symbol that take care of whole steps involved in a reduce action. The selected symbol sends a kill message to its neighbors next, them and itself are converted on a new process corresponding to the rule's LHS. This new process should have its behaviour just for continue the parsing process according to the grammar rules. Therefore, some symbols there will have responses to kill action and some code to wait until new message come into its communication channels. Of course, for the design of this symbols net we take into account the same information of shift-reduce parsing.
We will represent each symbol by a node of a process net. Such a process will have two channels to communicate with its neighbors. Let us consider an input string which is converted to a chain of process connected through its channels. Thus, we suppose that the channels at the chain boundary of a given string net are defined by special symbols, say `#` and `$`.

$$\ldots PQR \ldots$$

Accordingly with the message in its two communication channels, the process realizes only one of the following actions:

1. Q is renewed and maintains a new state: PQR.
2. Q becomes a new symbol S: PSR.
3. Q is dead and shortcircuit: PR.
4. Q is able to shortcircuit with its neighbors and becomes a new symbol U

Usually the terminal symbols die inmediatly given place to its grammatical category. There are two main actions for non-terminal symbols:

1. It will wait a message to die
2. It will be able to "head" of some string RHS grammar rule

In the former the process which represents the symbol will be suspended until certain message take place. The later is similar, but the corresponding process will send messages to its neighbors, yielding the state of these.

3.2 The SymAct language

The language which supports the above ideas, called SymAct, is compiled into Concurrent Prolog. For SymAct language we have formulate a simple syntax. An actor symbol description consists of a set of clauses. Each clause follows the shift-reduce information of that symbol, and each clause has the form:

```
head :
       guard1→ body1.
       guard2→ body2.
          ...
```

where the `head` will be formed by a name, the grammar symbol and its parameters, which in this case are constrained to show the communication channels. On the `guard` part we will have the condition provided by the shift-reduce information to activate the corresponding `body`. Let us show some examples for clearing this ideas.
Given a production $A \rightarrow b$ we look at the information about the grammar symbols: b followed by `$` is reduced to A, and A followed by `$` terminates the parsing, accepting the string. Additionaly we include the left delimiter `#`. Thus we have:

```
b(_ , $) :
      true→ A(_ , $).
A(# , $) :
       true→.
```

At the first clause we have the die of the `b` actor and the creation of a new actor `A` with the same channels. At the second clause the actor is destroyed because of the condition at its channels. We use the `_` symbol to represent a don't care stream. Obviously, the `true` guard is always verified. In the SymAct description we will use the concatenation `Q_` to represent a message stream whose top message is `Q`. Let us consider a grammar with a recursion rule:

$$A \rightarrow aA$$

$$A \rightarrow a$$

We would have the next actor description:

```
a(_ , Right) :
      Right==$→ A(_ , $)
      Right==A_→ Kill A, A(_ , _)
      true→ DiscardR, Wait
A(Left , Right) :
      Left==Kill→ Shortcircuit
      Left==#,Right==$→ Print(accept)
      true→ DiscardL, Wait
```

As we can see, at the actor description head, the communications channels denote the message for income to the symbol. Here we are using several new keywords more. One of this has as prefix `Discard`, and it means that will be discarded the symbol read at the right channel if it is followed by `R` or to discard the symbol read at the left channel if it is followed by `L`. The `Wait` keyword indicates to the head actor to stay active until a new message arrives to some channel. Particularly, at the actor symbol a there are two guards which create new processes or send messages to its neighbors. In the first case whenever is being covered the shift-reduce conditions, we have a *becomes* instruction, changing the orignal actor symbol by a new one. The second case corresponds to the service of a special message from the right neighbor, it is send a message for kill the right neighbor and newly the actor symbol is changed because it represents a reduce case. At

the last case there is not guard condition and the actor symbol only cleans its right mailbox and wait for a new message. By a similar way we can explain the actor symbol A. It would be only necessary to say the meaning of `Shortcircuit` instruction. This is excecuted when the actor symbol decides to die, but its neighbors need to continue receiving messages; the left neighbor needs the messages from the right mailbox and the right neighbor needs the messages from the left mailbox. Thus the actor symbol connects the left communication channel with the right communication channel through a `Shortcircuit` instruction. We note that SymAct inherits some aspects of parallel logic programming. Here `Right` is a variable representing the right channel whose content is expected to be a stream. If the first guard of `a` is satisfied then the whole stream was instantiated as only one symbol.
The next example shows how to deal with ambiguity. Let a simplified grammar to describe some cases (*genitum, accusare* and *ablatum*) of Spanish noun presentation be:

$$\begin{array}{lcl} Orac & \rightarrow & Verb\,Comp \\ Comp & \rightarrow & Acus\,Ablat \\ Acus & \rightarrow & Fn\,Gen \\ Gen & \rightarrow & \lambda|\text{de}\,Fn \\ Ablat & \rightarrow & \lambda|Prep\,Fn \\ Fn & \rightarrow & Sus|Det\,Sus \end{array}$$

With these rules we can have the sentence:

Llenó la bolsa de dinero

which is ambiguous. It follows the SymAct clauses for the above grammar:

```
Verb(# , Right_) :
     Right==Comp  -->
         Kill Comp, Orac(# , _).
     true  -->
         DiscardR, Wait.
de(_ , Right_):
     Right==Fn  -->
         G=SkipR(Fn),Gen(_,G),Prep(_,Right_).
Fn(_ , Right_) :
     Right==Gen  --> Kill Gen, Acus(_,_).
     Right==Ablat,Right==$  -->
        Acus(_,Right_).
Acus(_ , Right_):
     Right==Ablat  --> Kill Ablat, Comp(_,_).
     Right==$  --> Comp(_,Right_).
Prep(_ , Right_) :
     Right==Fn  --> Kill Fn, Ablat(_,_).
Sus(Left_ , _) :
     Left==Det  --> Kill Det, Fn(_,_).
     true  --> Fn(L_,_).
```

Some of these clauses need to pay attention to an eventual kill message. Then, in SymAct we may add the next generic kill-condition:

```
*(Left_ , Right_) :
    Left==kill→ DiscardL, ShortCircuit.
    Right==kill→ DiscardR, ShortCircuit.
```

At `de` clause we have the creation of two new actors which correspond to the two possiblities of bootom-up syntax tree construction, because of the ambiguous grammar rules.

3.3 Some compiled examples

Compiling actor symbols should consider some additional items to the previous description. First of all, there will be two channels more which represent the output of the process to its neighbors. Also it is necessary instantiate the corresponding variables with the adequate messages that is going to send. Let us see the first example. We have the next three clauses:

```
b(X, b|Y, b|Z, ?W1|W) :- W1==$ |
 Y=A|Y1, W=A|W2,
 A(X, Y, Z, W).
b(X, Y, Z, ?W1|W)     :- W1=/=$ |
 b(X, Y, Z, ?W).

A(?X, Y, Z, ?W)       :- W==$, X==# |
 print(accept).
```

The first clause is verifying the reduction of b to A, then the guard checks this condition and sends messages to its neighbors the new name of it. Finally it creates the process A. The next clause waits until income a message. Whenever the message be "don't care" for b this is discarded and the process becomes itself. The last clause reflects the accepting condition: A preceded by `#` and followed by `$`. Then the symbol waits for those conditions and resume.
Now, let us observe the example of recursion:

```
(a2)  a( X1 (a|X2) (a|X3) X4) :- ?X4==$:
         X2=(A|5),
         X3=(A|6),
         A( ?X1 X2 X3 X4).
(a3)  a( X1 (a|X2) (a|X3) X4) :- ?X4==(A|6):
         X2=(A|X7),
         X3=(k A|X8),
         A( ?X1 X2 X3 ?X6).
(a1)  a( X1 X2 X3 X4) :- ?X4==(X5|X6):
         a( X1 X2 X3 ?X6).

(A1)  A( X1 (A|X2) (A|X3) X4) :- ?X1==(k|X6):
         X2=X4,
         X3=X6.
(A3)  A( X1 X2 X3 X4) :- ?X1==#, ?X4==$:
         print(accepted).
(A2)  A( X1 X2 X3 X4) :- ?X1==(a|X8):
         A( ?X8 X2 X3 X4).
```

In this program we have a clause to clean the input channel (a1, A2), clauses to reduce according to first production grammar (a3), clauses to reduce according to the second production grammar (a2) and to accept the string (A3). The (A1) clause is to reply to the destruction message. We can observe at (a3) clause two messages to the right neighbor: the kill message and the name of the new process. Furthermore, in (A1) the process indicates the shhortcircuit of its communications channels.
When we have a state such that contains at least two shift-reduce conditions we have reach to an ambiguity state. The processes associated with those symbols should represent this situation. Let us see a simple example of this case. In the next production rules represent an ambiguous language:

$$B \rightarrow A$$

$$B \rightarrow AC$$

$$A \rightarrow a$$

$$A \to aC$$

$$C \to bc$$

If we like to take into account all paths created for the syntactic tree related to a string in the language, we must consider all possible reduction at any point of each production. Let us consider the rules for A, we would have the next clause:

```
a( X, Y, Z, W) :- ?W==[C|W1] :
  A( X, Y, Z, W),
  A( X, Y, Z, W1).
```

Here there is a reduction from symbol a to A and the case for postpone the reduction through the second predicate body.

4 Conclusions

We have presented a language which permits the specification of actions performed during the senteces parsing in an actor fashion. Until now we have implemented SymAct programs in the Concurrent Prolog virtual machine. With the tested grammars we have found SymAct well situed to deal ambiguous grammars. However, the present version would appear some inefficient because of the message traffic overhead. But it is important to remark that the power of SymAct is just the communication of properties between symbols. Through symbol communication is why its creation motivated. Thus when we process long discourses often it is necessary communicate piece discourse atributes like topic, referents, and others. This type of tests have not been realized yet, but actually this is the next step in the development of SymAct.

References

[1] Agha, Gul, Foundational Issues in Concurrent Programming, *SIGPLAN notices*, Vol. 24 (4), 1989, 60-65.

[2] Aho, Ullman and Sethi, *Principles of Compiler Disign*, Addison Wesley, 1986.

[3] Foster, Ian and Taylor Stephen, Strand: A practical parallel programming language, *Proceedings of the North American Logic Programming Conf.* 1989.

[4] Kahn, Kenneth, Tribble, E. Miller, M. and Bobrow, B., Vulcan: Logic Concurrent Objects, *Research Directions in Object-Oriented Programming*, MIT Press, 1987, 75-112.

[5] Kahn, Kenneth and Saraswat, Vijay, Actors as a Special Case of Concurrent Constraint (Logic) Programming, *OOPSLA ECOOP '90 Proceedings*, 1990, 57-66.

[6] Matsumoto, Y., A Parallel Parsing System for Natural Language Analysis, *3rd. Int. Conf. of Logic Programming*, (E. Shapiro, ed.), Springer-Verlag, 1986, 396-409.

[7] Saraswat, Vijay, *Concurrent constraint programming languages*, MIT Press, 1990.

[8] Simpkins, Neil, Chart Parsing in Prolog, *New Generation Computing*, (8), 1990, 113-138.

[9] Takeuchi, Akikazu, *Parallel Logic Languages*, John Wiley & Sons, 1992.

[10] Tomita, *Generalized Botom-up Parsing*, Kluwer Press, 1989.

[11] Wegner, Peter, Concepts and Paradigms of Object-Oriented Programming, *OOPS Messenger*, Vol. 1 (1), 1990, 7-87.